MSF Build System

Lorenzo Flückiger

Developers Documentation

Table of Contents

Requirements

The MSF build system uses the standard gmake program available on all MSF target platforms: SGI-IRIX, Sun-SunOS, Intel-Linux and Intel-Windows[1].

How to use

The MSF build system is based on a set of partial makefiles defining variables and rules that allow the developer to benefit from a powerful build system with a minimum of effort. All these makefiles are located in $MSF_HOME/makes.

The principle of the MSF build system is the following: the developer defines a set of variables which will be the input of the predefined rules as well as the final targets (libraries and/or executable). By including the top level partial makefile, most of the work is automagically done.

The simple example illustrated on Figure 1 demonstrates a typical usage of the MSF build system.

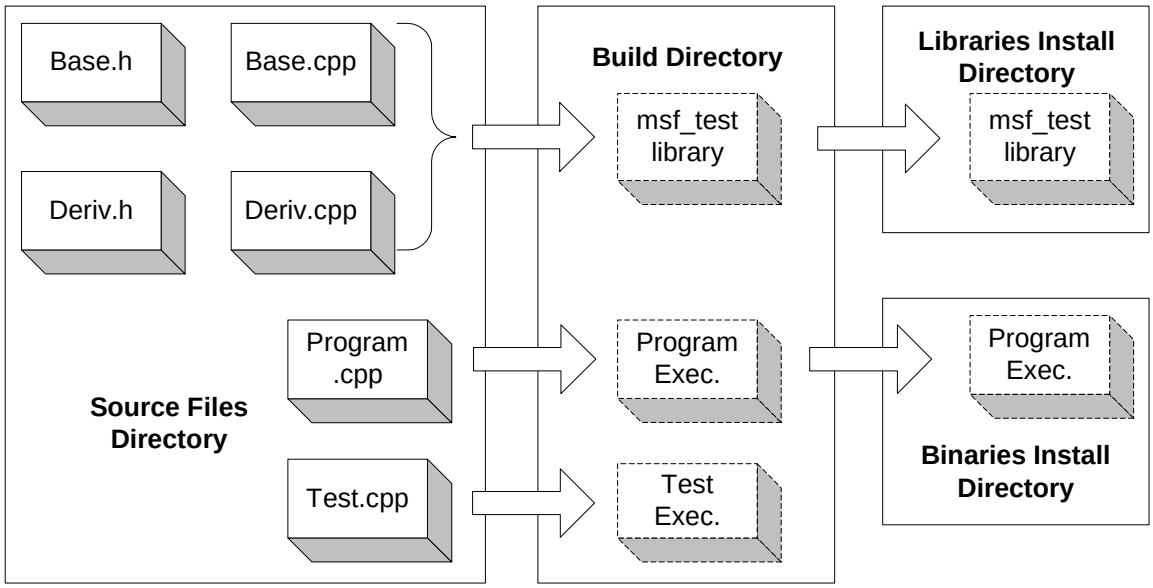

Figure 1: Build example of a simple set of source files

Consider the four files `Base.h`, `Base.cpp`, `Deriv.h`, `Deriv.cpp` that will form a library named `msf_test`, which is part of the MSF standard libraries. The program `Program.cpp` will become an executable which is also one of the

1. The Cygwin package is needed to build MSF under Window, even if the Visual-C++ compiler is used.

standard programs provided by MSF. Also, the directory contains a `Test.cpp` program which is used only locally.

The first step is to define the input variables that will control the build system (see Section “Variables used by the MSF build system” for a full description of these variables):

```
LIB_SRC := Base.cpp Deriv.cpp # source file(s) for the library

LIB_NAME := msf_test          # library name

EXEC_SRC := Program.cpp       # MSF program source file(s)
TEST_SRC := Test.cpp          # Test program source file(s)

LIBS_TO_INSTALL := msf_test   # library(ies) to install
PROGS_TO_INSTALL := Program   # executrable(s) to install
```

Then additional flags for the compilation / link can be specified by adding them to the list of the standard variables:

```
LDLIBS += -lmsf_test          # Libraries to link with
LDFLAGS += $(MSF_LIB_DIR)     # Path to these libraries
```

The key point is then to include the top level makefile of the MSF build system[1] (located into the `$MSF_HOME/makes` directory):

```
include ../make.incl
```

At this point, several new variables have been automatically generated. they can be used in the remaining portion of the makefile (examples are given for a Linux build, with `MSF_HOME` set to `/projects/MSF`).

- `TEST_EXE` contains the name of all the test programs: `Linux/Test`
- `EXEC_EXE` contains the name of all executables: `Linux/Program`
- `MSF_PROGS` contains the target destination names of the programs to install: `/projects/MSF/bin/Linux/Program`
- `MSF_LIBS` contain the target destination names of the libraries to install: `/projects/MSF/lib/Linux/libmsf_test.a`

These variables are a convenience to define what the makefile has to produce. For example, simply specifying the following rule:

```
default: $(MSF_LIBS) $(MSF_PROGS) $(TEST_EXE)
```

will compile and install the libraries and programs. This will be achieved by following the MSF rules which define how to generate the dependencies, how to compile, how to link and how to install files. Note the `TEST_SRC` is a separate variable allowing to exclude test programs from the `MSF_PROGS` installable

1.Note that the MSF build system require the input variable to be defined before including `make.incl`.

programs. To also build the test programs, TEST_EXE have been included in the default rule.

Note the MSF build system requires a default rule to work correctly (the first target which is called build_goal inside make.common simply calls the default target). If the default rule is not present in the user makefile, a message like:
"no rule to make target 'default' needed by 'build_goal'" will be issued.

In addition to the rule specifying the main target, an additional dependency rule will probably be needed:
$(EXEC_EXE): $(LOADLIBES)
It specifies that the program files to be produced are depending on the library composed of the LIB_SRC objects files (and then will be linked against).

At this point, simply typing make on the command line will build everything in the current directory. This example makefile is provided in the MSF_HOME/makes/tests directory.

Configuration

ENVIRONMENT VARIABLES

The MSF build system uses several environment variables that have to be defined. The user is free to define these variables using his preferred method, however, a file called msfsource is provided and can be configured to reflect the current setup.

MSF_HOME Defines the top directory of the current MSF tree

RTI_HOME & RTI_BUILD_TYPE Defines the location of the RTI to use

FLTK_HOME Define the location of the FLTK package (only needed by the components requiring a GUI based on the Fast Light ToolKit)

In addition to these environment variables, a set of more fine tuning parameters can be adjusted in the platform dependant makefiles: make.Linux, make.IRIX, etc. There the compiler/linker can be reconfigured as well as the various options dependant on a particular platform. It should be noted that the compiler command to invoke can also be defined with the environment variable COMPILE_CMD, which case will override the one defined in the makefile system.

PLATFORM DEPEND BUILD

The MSF build system is configured to work on multiple platforms with the same source code: all dependencies, object files, libraries and programs are generated in sub-directories reflecting the platform name. By default MSF uses the convention described in Table 1.

However, when there is a need to mix multiple flavors of compiler/libraries versions on the same platform (like gcc-2.95 or gcc-3.1 under a Suse 8.1 Linux), more characterized directory names are required. The MSF build system supports this by setting the environment variable MSF_USES_FULL_ARCH. For example with a

Platform	Sub-directory name
Any Linux on Intel processors	`Linux`
Any IRIX on MIPS processors	`IRIX`
WindowsNT, Windows2000	`Win32`

Table 1: Platform dependant directory names

cshell:
`setenv MSF_USES_FULL_ARCH 1`

In this latter case, one will have to create into the makes directory a makefile reflecting the particular architecture (it also can be simply linked to one of the default platform dependant makefile file). For example if `MSF_USES_FULL_ARCH` is used on a Linux system running on an Pentium processor and using gcc version 2.95 and glibc 2.2, a configuration file named `make.ix86-linux-gcc2.95-glibc2.2` will be required.

The architecture dependant names are generated with a shell script archname.sh located in the MSF_HOME/makes directory.

`archname.sh` Return the architecture name from queries to the system/compiler and is used to generate platform dependent directories. The modifier -a can be added if fully qualified architecture names are required. Do archname.sh -h to see all the options.

BUILD CONFIGURATION The build configuration regarding optimization and type of libraries is also set in the platform dependent makefiles. They define a variable `BUILD_CONFIG` which contains the characteristics of the desired build. Currently only two options are available:

- Optimized or Debug version set with `optimized` and `debug`
- Dynamic or Shared libraries set with `dynamic` and `static`

For example the following definition in `make.Linux`:
`BUILD_CONFIG := debug static`
will build MSF using static libraries with debug information.

How it works

The MSF build system is based on a set of partial makefiles having distinct roles:

`make.incl` Is the top level makefile to include from the working makefile. This make portion mainly does some sanity checking and includes all the other necessary makefiles.

`make.`*`arch_name`* Defines a set of variables (compiler name, file extensions or compile flags) that are platform dependent. This file can be modified to meet specific platform needs.

make.dirs Note: make.dir is not used any more in the current version of MSF. Some of the important directories definition are simply put in the platform dependant makefiles. Defines all the directories used by the MSF build system. This makefile uses the variables `MSF_HOME, RTI_HOME, RTI_BUILD` and `FLTK_HOME` and can be included by external project makefiles for convenience.

make.rules Is the core of the build system. It expands input variables into complete filenames and contains all the rules to make the various targets.

make.common Contains some common rules like how to `clean` directories or `list` the files. This make file is not included by default and it is to the user to insert it into his own makefile if he would like to use it (which is strongly encouraged). As `make.common` contains mainly `PHONY` targets, it should be included at the end of the user makefile: `include ../make.common`

make.docs Contains instructions relative to the documentation generation process.

As mentioned, the behavior of the build system is really determined by `make.rules`. Figure 2 shows how the rules cascade from initial targets during the build process.

Variables used by the MSF build system

To help MSF developers to keep simple makefiles, the MSF main make system uses a set of variables as inputs for its rules.

INPUT VARIABLES

LIB_SRC List of sources files needed to build the library.

LIB_NAME Name of the library to be generated. This name is the base name of the library. The makefile system will expending it according the build platform and configuration. For example, if `LIB_NAME` is set to `test`, the full library name under Linux when building shared libraries will become: `libtest.so`.

LIB_PATH Optional path where to put the local library. This is a modifier for the LOADLIBES generated variable. By default the library goes in a platform dependant sub-directory from the current makefile path. By defining LIB_PATH, it is possible to redirect this library somewhere else. This is particularly useful when multiple directories with their own makefile all participate in building the same library.

EXEC_SRC List of source files that will generate executables intended to become part of MSF distributable. To actually have these executable installed, you will have to list the program names into `PROGS_TO_INSTALL` and define `MSF_PROGS` has one of the makefile target. Note that without defining `PROGS_TO_INSTALL, EXEC_SRC` can be used the same way than `TEST_SRC`.

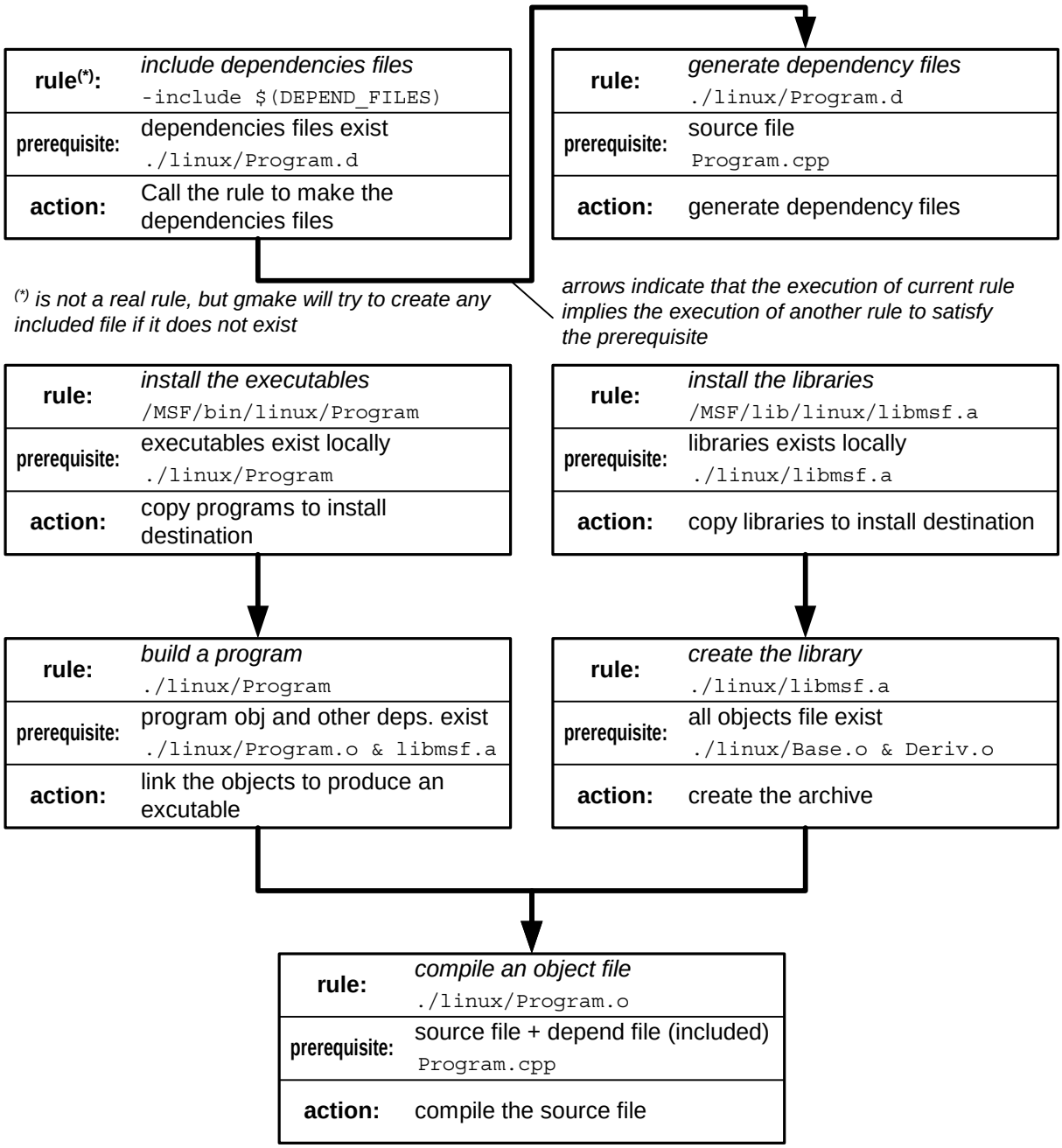

Figure 2: Propagation of the rules in the MSF build system

TEST_SRC List of source files that will generate test programs. TEST_SRC could be use even if some programs are not test programs, but are designed to remain local to the working directory.

LIBS_TO_INSTALL A set of libraries name. The libraries listed here will be build and copied into the INSTALL_LIB_DIR (defined in make.dirs) directory. The libraries names follow the same rule than LIB_NAME.

PROGS_TO_INSTALL A set of programs to install. The programs listed here will be build and copied into the INSTALL_BIN_DIR (defined in make.dirs) directory. The program names should not contain any extension which will be added automatically by the makefile according to the platform, For example, if PROGS_TO_INSTALL is set to MyProgram, then under Windows the program name will become MyProgram.exe.

From these variables, a set of generated variables are produced by the makefile. Some of these variables are for internal makefile use mainly, while others will be used by the component developer in his makefile.

GENERATED VARIABLES

LIB_OBJS List of object files composing the library (normally not used in your makefile). The LIBO_OBJS files are composed with the build architecture directory and the correct extension. For example File.cpp becomes Linux/File.o.

EXEC_OBJS List of object files corresponding to the MSF executables. Same rules as for LIB_OBJS applies.

TEST_OBJS List of object files corresponding to the test programs. Same rules as for LIB_OBJS applies.

EXEC_EXE List of produced MSF programs. The generated names contain the output directory and the correct file extension (.exe under Windows)

TEST_EXE List of produced test programs. Same rules as for EXEC_EXE applies.

LOADLIBES The full name of the library to be produced (including optional LIB_PATH, build architecture, library prefix and library suffix):
mylib -> Linux/libmylib.so.

MSF_LIBS List of libraries to install with the full path to the output directory.

MSF_PROGS List of the program to install with the full path to the output directory.

DEPEND_FILES A list of all the dependency files: it is generated from LIB_SRC, TEST_SRC and EXEC_SRC.

STANDARD VARIABLES

The MSF build system uses the standard make conventions for the variables used in compile and link rules. MSF makefiles only add arguments to these variables, meaning that one can add parameters by setting the corresponding environment variable. For example, if the developer wants a special build for testing purpose with the preprocessor argument “-DMY_TEST”, it can do so without modifying the

makefile by setting it in the shell:
`setenv CPPFLAGS -DMY_TEST`

The following standard variables are used and can then be augmented with environment variables:

CXX The compiler to use. This is the only environment variable which is taken as is (not added to the makefile one) if it is defined. For example, under Linux, the architecture makefile defines the compiler to be g++. If one wants to test with a another compiler, it could do:
`setenv CXX /opt/experimental/gcc`
(Note that this example is a bad idea of how to use another compiler: it will be better to have the PATH and LD_LIBRARY_PATH variable set correctly for an alternate compiler)

CPPGFLAGS The preprocessor flags (for example `-DNDEBUG`)

INCLUDES Directive for files search used by the preprocessor (for example `-I../GUI`)

CXXFLAGS The compiler flags (for example `-g`)

LDFLAGS Flags for the linker (for example `-L../GUI/Win32`)

LDLIBS Libraries to link with (for example `-lmygui`)

Dependencies

The dependencies are generated automatically by the make process. There is no need to specify explicitly `make depend` each time a change in the dependency has occurred. The dependencies is updated each time it is needed.

The behavior uses the makefile remake capability of gmake (see GNU make manual, section 4.12: Generating Prerequisites Automatically): for each source file (`file.cpp`) a corresponding dependency file (`$(OS)/file.d`) is generated. The dependency file `file.d` specifies the dependencies of the object file (`file.o`) on all the source files required, and includes in addition a dependency of the file itself (`file.d`) on the same set of source files to ensure that the dependencies are up to date. Finally the list of dependency files is simply included in the top makefile (with some additional tests to avoid including dependencies for special targets like `clean` our `count`):

```
-include $(DEPEND_FILES)
```

A typical dependency file will look like this (e.g. `Linux/Base.d`):

```
Linux/Base.d Linux/Base.o: Base.cpp Base.h
```

Libraries naming convention for linking search and prerequisites expression

The libraries are expressed in the MSF build system with the form `-lmylib`. It allows the linker to search for libraries names `libmylib.so`, `libmylib.so.1` or `libmylib.a`[1]. In addition, it allows to define the same compact library name on all platforms, without worrying about the extensions.

But most important, the same -lmylib syntax can be used in the prerequisite of rules. For example, if a program `Program.exe` depends on the library `libmylib.lib` the dependency could be expressed like:

```
Program.exe: Proram.cpp -lmylib
```

(Note that this syntax is not platform portable, but just here for illustration)

To allow this behavior, the build system expands the libraries names with their full path by searching for these libraries in a set of directories. The MSF build system uses the `vpath` command to instruct makefile to search for MSF build libraries in their install directory.

If a developer need to add directories to the search path because some of his libraries are not located in the standard MSF libraries directory, he can simply add arguments to the `VPATH` variable. For example:

```
VPATH += ../GUI/$(OS)
```

Automatic Windows/Unix filenames handling

MSF build system requires Cygwin under Windows to take advantage of the makefile program and several Unix utilities (sed, awk, uname, etc). In addition, MSF currently also uses gcc under Windows to create the dependencies. However, MSF uses the Microsoft Visual C++ compiler/linker. This raises a pathname problem:

- makefile and gcc -MM (dependencies) use and produce Unix like filenames (for example `/cygdrive/c/Users/My\ Msf/Makefile`)
- cl.exe and link.exe require Windows filenames (for example "`C:\Users\My Msf\Makefile`")

The problem is solved by the MSF build system using the `cygpath` utility when defining the build rules:

- The dependency rule convert filenames starting with MSF_HOME to their Unix counterpart
- The compile/link rule convert filenames starting with MSF_HOME to their Windows counterpart

1. Under Windows the libraries makefile will search for likbmylib.lib and libmylib.dll

These integrated build rules free the user of bad path names problems and let him use either Unix or Windows path in the environment variable definition of MSF_HOME. Note that the definition of other paths like RTI_HOME should use the Windows convention because these path are not converted in the compile/link rule. However, libraries external to MSF are not used in the prerequisite generation, and the Windows path name does not hurt the build system.